Objective as a Feature for Robust Search Strategies

Anthony Palmieri, Guillaume Perez

Objective as a Feature for Robust Search Strategies

Anthony Palmieri[1,2] and Guillaume Perez[3]

[1]Huawei Technologies Ltd, French Research Center
[2]Université de Caen - Normandie
GREYC
[3]Cornell University, Department of Computer Science
Ithaca NY 14850, USA
anthony.palmieri@huawei.com, guillaume.perez06@gmail.com

Abstract. In constraint programming the search strategy entirely guides the solving process, and drastically affects the running time for solving particular problem instances. Many features have been defined so far for the design of efficient and robust search strategies, such as variables' domains, constraint graph, or even the constraints triggering fails. In this paper, we propose to use the objective functions of constraint optimization problems as a feature to guide search strategies. We define an objective-based function, to monitor the objective bounds modifications and to extract information. This function is the main feature to design a new variable selection heuristic, whose results validate human intuitions about the objective modifications. Finally, we introduce a simple but efficient combination of features, to incorporate the objective in the state-of-the-art search strategies. We illustrate this new method by testing it on several classic optimization problems, showing that the new feature often yields to a better running time and finds better solutions in the given time.

1 Introduction

Solving combinatorial optimization problems is known to be a hard task, but constraint programming (CP) enables tackling several of them [26, 22]. One of the CP strength leans on an efficient search for a solution in the variables' domain space. The resolution of industrial problems often relies on dedicated knowledge experts to build a good search strategy (SS) [25, 23]. But such information, while appealing, is not always available nor possible. That is one of the main motivations for the development of black-box constraint solvers, where the only user's concern is to build an efficient model. Black-box solvers need robust and efficient SSs, and many researches have been done [11, 18, 5, 16, 28]. Notably, in Constraint Programming, activity-based search (ABS) [14], impact-based search (IBS) [20] and weighted degrees (Wdeg) [1] are well known state-of-the-art search strategies for combinatorial problems.

In CP a search heuristic usually consists of choosing a pair $(variable, value)$, called a decision. Then, a binary search tree is built to explore the search space. The solving time is highly correlated with the size of the search tree. A shorter run time is usually expected when a smaller tree is explored. Since the search strategy determines how to build the search tree, the solving time is strongly impacted by the search strategy, and can differ by order of magnitude.

Most search strategies use the first fail principle [7, 24]. This principle tries to fail as soon as possible, in order to reduce the search tree size. The first fail principle is very efficient in practice for constraint satisfaction problems (CSPs), for example SSs such as IBS and ABS consider the variables' domains as feature to make decisions, and try to find variables having the potential to reduce the other variables domains, while WDeg uses fail counters and the constraint graph.

A constraint optimization problem (COP) can be seen as a CSP with an objective function to optimize. Solvers often have a variable representing the possible values of the objective function. When a new solution is found, a constraint is added, requiring the next one to be better. Once the best solution is found, the next step is to prove its optimality. In other words, solving a COP includes: finding the best solution, and proving that no better solution exists. In practice, it is unknown whether the current solution is the best one until the exploration of the search tree is completed.

Search strategies are mostly designed to reduce the search space by focusing on constraint satisfaction, but with COPs, the objective value can additionally be used to reduce the search space. Two different solutions might prune the search space, depending on their respective objective values. In COP, finding a good solution can drastically reduce the search space, by avoiding the exploration of less promising parts of the search tree, with respect to the objective. This implies that the order in which solutions are found has a strong impact on run-time for the complete space exploration, unlike for CSPs.

This observation is one of the main motivations of this paper. Our idea, inspired by integer programming [4], is to extract good features from the objective variable to make good decisions. A recent CP work started exploring this area by using the objective to design a value selector in order to find a first good solution [3]. An advantage of using the objective as a feature is its ability to both optimize the objective value and to reduce the search tree at the same time. Such information, as will be shown in the experimental section, can drastically help SSs to make better decisions.

In the following, we design a function (Δ_O) monitoring the objective bounds modifications along the solving process. This function is one possible implementation of an objective-based feature extractor ($\widetilde{\Delta_O}$). We then define a variable selector based only on $\widetilde{\Delta_O}$, named Objective-Based Selector (OBS), which selects the variable maximizing $\widetilde{\Delta_O}$. Lastly, in order to take advantage of this new objective feature and the many existing ones, we propose a simple but efficient hybrid method to combine search strategies, such as IBS, ABS etc, to take into account the newly introduced objective feature. Finally, we show the efficacy of these new hybrid strategies compared to the original strategies on all the optimization problems from the Minizinc challenge library [15].

2 Preliminaries

2.1 Constraint Satisfaction Problem (CSP)

A *CSP* is a pair $P = (X, C)$ where $X = \{x_1, x_2, ..., x_n\}$ is a set of variables and $C = \{C_1, C_2, ..., C_m\}$ is a set of constraints. A variable x_i is associated with a domain $D(x_i)$, representing all of its possible values. A constraint C_i contains a set of all its allowed tuples defined over a subset $S_{C_i} \subseteq X$ of variables.

A solution is a tuple of values $(a_1, a_2, ..., a_n)$ such that the assignments $x_1 = a_1$, $x_2 = a_2$ ..., $x_n = a_n$ respect all the constraints. The solving process for a CSP generally involves a depth first search algorithm with backtracks in a decision tree. At each node of the tree, a propagation algorithm is run, which iteratively uses a dedicated filtering algorithm to check the validity of each constraint. Each filtering algorithm reduces the search space by removing the values that cannot belong to a solution. Finally, a solution is found when all variables are instantiated to a value.

A constraint optimization problem (*COP*) is a pair (P, F_O), where P is a *CSP* and F_O is an objective function that has to be optimized. Without loss of generality, we consider here only minimization problems. All solutions to a COP are not equivalent, as their overall quality is determined by the objective value $F_O(sol)$. The solving process of a COP is analogous to a CSP, except that it contains an objective constraint. This constraint ensures that the next solution found will be better. This paper aims to use this constraint in order to reduce the search space.

2.2 Search Strategies

A search strategy (SS) for constraint programming (CP) determines how the search tree is built during the solving process. At each node of the search tree, the SS chooses a non-assigned variable and a value belonging to its domain. A decision often corresponds to a pair *(variable, value)* which can be seen as a backtrackable constraint *variable = value*. Search strategies are crucially important to find good solutions, to reduce the search space, and even to quickly find an initial feasible or good solution [3].

We briefly describe three state-of-the-art SSs. For a more complete description please refer to their original publications.

Impact Based Search (IBS) [20] selects the variable whose choice is expected to provide the largest search space reduction. To do so, *IBS* considers the cardinality reduction of the Cartesian product of the domains (called the impact). Thus the main feature of this SS uses variables' domains.

More formally, let x be a variable, and v be a value belonging to the current domain $D(x)$. Let P_{before} (resp. P_{after}) be the cardinality of the Cartesian product of the domains before (resp. after) the application of the decision $x = v$. The impact of a decision is:

$$I(x = v) = 1 - \frac{P_{after}}{P_{before}}$$

Let $\bar{I}(x = v)$ be the average impact of the decision $x = v$. Then, this impact of a variable x with current domain $D(x)$ is computed by the following formula:

$$\bar{I}_x = \sum_{v \in D_x} 1 - \bar{I}(x = v)$$

At each node the free variable having the largest impact is assigned to its value having the smallest impact. Note that this search is an adaptation of pseudo-cost-based search from mixed integer programming.

Activity Based Search (ABS) [14] selects the most active variable per domain value. A variable's activity is measured by counting how often its domain is reduced during the search. Thus, once again, the feature of this SS uses the domains of the variables. More formally, the number of modified variables is monitored and stored in $A(x)$, which is updated after each decision with the following rule:

$$\forall x \in X s.t. |D(x)| > 1 : A(x) = A(x) \times \gamma$$
$$\forall x \in X_0 : A(x) = A(x) + 1$$

X_0 represents the set of variables reduced by the decision and $\gamma \in [0, 1]$ is the decay parameter. *ABS* maintains an exponential moving average of activities by variables' value. At each node, *ABS* selects the variable with the highest activity and the value with the least activity.

Weighted Degree (WDeg) [1] uses the constraint graph to make decisions. *WDeg* counts the number of failures ω_c for each constraint c . *WDeg* features are the constraint graph and the fail counters. *WDeg* first computes, for each variable x, the value $wdeg(x)$, which is the weighted (ω) sum of the constraints involving at least one non-assigned variable. *WDeg* then, selects the variable having the highest ratio $\frac{|D(x)|}{wdeg(x)}$.

3 Objective Function and Search Strategy

Search strategies aim to reduce the search space, but additionally aim to find good solutions as quickly as possible. Most SSs choose the hardest variables to satisfy first, the main challenge being to find such variables. While most SSs decisions were based on variables domains, the constraint graph, etc, objective-value based decisions are rarely done in CP. One of the reasons is that, in CP, we cannot easily back-propagate the objective to the variables to make decisions as done in Mixed Integer Programming. But even if we can not have such exact information, not taking into account the variables impacting the objective value can lead to an exponential loss in time. This is shown by the following synthetic example.

Example Consider a COP having $n+m$ variables and whose objective is the sum of the last m variables. This problem has an AllDifferent constraint [21] over all the variables. Ignoring the objective value can lead to the search tree shown in Figure 1 (left). In this example, the strategy focuses only on other features, without taking the objective into account. Whereas a strategy that considers the objective detects variables having high impact on the objective, and consider them earlier enabling a potential reduction of the search tree.

Moreover, we can find high quality solutions earlier and these solutions prune the search space more efficiently. As we can see, the processing of the m variables is repeated an exponential number of times (d^n). This is because the variables that impact the objective are chosen too late leading to a bigger search tree.

The search tree using the objective value as a feature is shown in Figure 1 (right). The last m variables are selected higher in the search tree, yielding better solutions faster and allowing to close the search using the objective sooner. Finally, by using the objective value, we obtain a smaller search tree.

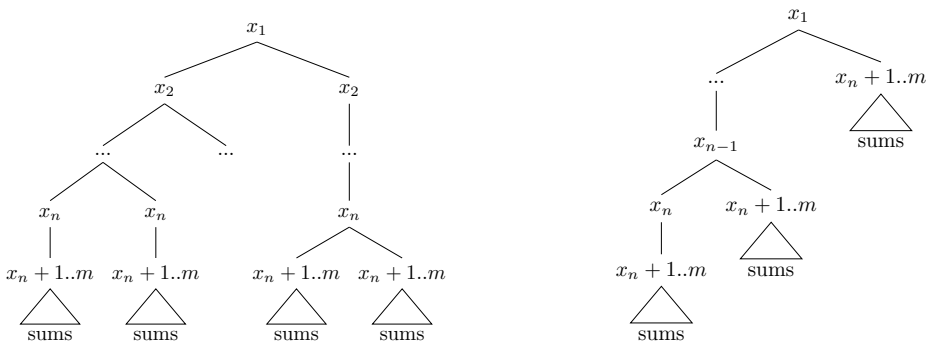

Fig. 1. (left) A search failing to consider the objective value. (right) An objective based search.

This simple example shows that the objective value allows to assign variables having high influence on the objective earlier and thus can help the solver to avoid considering useless parts of the tree. The idea is to consider as soon as possible the variables impacting the objective. We now define a new feature based on the objective, which we will use to define an objective-based search strategy.

3.1 Objective modifications as a feature

The proposed feature focuses on the objective bounds modifications by using a function Δ_O. The upper and lower bounds are separately considered as two different pieces of information. Let O be the objective variable to optimize. Let s and $s-1$ be respectively the current and the previous node of the search tree. Let $\overline{\Delta_O}$ (resp. $\underline{\Delta_O}$) be the upper (resp. lower) bounds difference between its value before and after the decision propagation. The function is defined as follows:

$$\Delta_O(s) = a \times \underline{\Delta_O} + b \times \overline{\Delta_O}$$

We choose to consider the upper and lower bounds separately. The choice of the parameters a and b defines the function behavior. The coefficients can take any value and correspond to the importance (positive or negative) given to each bound. For instance, in minimization problems, the coefficient a of lower bound modification corresponds to the weight for the consideration of removing the best potential solutions. While, the upper bound modification coefficient b, represents the weight to consider the deletion of the worst potential solutions.

Note that this function has a more fine-grained description of the objective than usual measures used in search strategies. Classic SSs monitor the modifications of the decision variables, but in general, treating differently the lower and upper bounds, has no meaning for such variables.

3.2 Objective-Based Selector (OBS)

We propose a new variable selector based on the Δ_O function: *OBS*. OBS first selects the variables having the highest impact on the objective with regard to the Δ_O function. To do so, the weighted sum of the Δ_O function values for each $x \in X$ is monitored through $\widetilde{\Delta_O}(x)$, and updated after each decision involving the variable x. The parameter γ is the degree of weighting decrease of the exponential moving average. The updated value $\widetilde{\Delta_O}'(x)$ is processed as follow:

$$\widetilde{\Delta_O}'(x) = \frac{\widetilde{\Delta_O}(x) * (1-\gamma) + \gamma * \Delta_O(x)}{\gamma}$$

At each decision, OBS selects the variable $x \in X$ such that $\forall y \in X, \widetilde{\Delta_O}(x) \geq \widetilde{\Delta_O}(y)$.

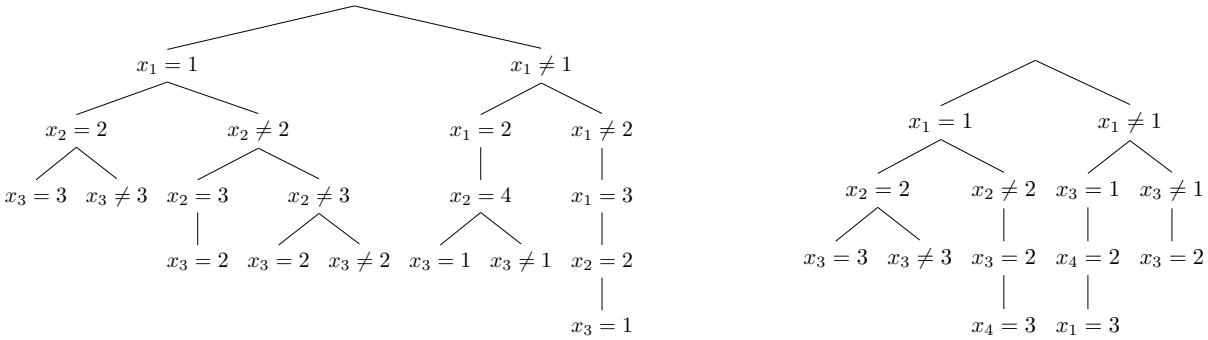

Fig. 2. Comparison of the search tree by a lexicographic search (left) and an objective based search (right)

Example Consider the didactic COP defined by the variables (x_1, x_2, x_3, x_4) having each as domain $D = [1, 4]$ and an *AllDifferent* constraint on the 4 variables. The COP's objective is $\min x_3 + x_4$. We use the parameters ($a = -1$,$b = 1$) for the Δ_O function, in order to penalize lower bound modifications and reward upper bound modification.

The tree search from Figure 2 shows the application of the objective based search strategy versus a lexicographic search. In this example, when a variable is selected, it is assigned to its minimum value. The lexicographic search on the left has more decisions than OBS on the right because it cannot identify which variable are important to satisfy the constraints and improve the objective.

At the beginning of the exploration, in the right tree showing *OBS* search, the variables x_1, x_2 and x_3 are selected and set to their minimum values. Each of these assignments has an effect on the objective's bounds and thus modifies Δ_O. When the decision $x_1 = 1$ is propagated, $\Delta_O(x_1)$ is set to -2 because of the changes of the objective domain from $[2, 8]$ to $[4, 8]$. The propagation of $x_2 = 2$ reduces the objective's domain

from $[4, 8]$ to $[6, 8]$ implying $\Delta_O(x_2) = -2$. When the variable x_3 is selected, the objective is instantiated to 7. This implies a $\Delta_O(x_3) = 0$. A solution is found with the value 7, so the next solution has to be smaller than 7. Afterwards the decision $x_3 = 3$ is refuted and the search tree is backtracked to the decision $x_2 = 2$ which is refuted, implying $x_2 \neq 2$. Then x_3 which has the highest Δ_O value is selected and instantiated to its domain's minimal value: 2. Then x_4 is the next free variable with the highest Δ_O value, 0. We thus select x_4 and assign it to its smallest value, 3. We find a solution equal to 5. Finally, when this branch is closed, the decision $x_1 = 1$ is refuted and by applying the OBS selection, the branch leading to the best solution is explored. An important aspect of this search is that it is close to human intuition to choose first variables x_3 and x_4 since they belong to the objective.

Note that the maintenance of $\widetilde{\Delta_O}$ values and the selection process are simple and not intrusive in solvers. Moreover, *OBS* does not need to change the constraints implementations.

3.3 Hybridization of search strategies

In this section we show how the objective and classical features can be combined together, to benefit from both. But most strategies should not be directly combined due to the range differences of their feature. For example, the *IBS* strategy has a value range between $[0, 1]$, while *ABS* one is between $[0, n]$. We propose to normalize all these values to fit in the interval $[0, 1]$ in order to combine them. Note that this applies to the Δ_O function as well.

Let $\widetilde{S^n}(x)$ be the normalized value of a search strategy S based on a classical feature. And let $\widetilde{\Delta_O^n}(x)$ be the normalized values for *OBS*. We combine the two pieces of information with the following formula:

$$S_O(x) = \alpha * \widetilde{S^n}(x) + (1 - \alpha) * \widetilde{\Delta_O^n}(x)$$

The hybrid search strategy selects the variable maximizing S_O. The values α and $(1-\alpha)$ represent the importance given to each feature. Note that α is in $[0, 1]$.

Example with ABS_O: While the *ABS* strategy uses the $\widetilde{A}$ values, storing the activities involved by the variables, our modification of the value associated with each variable is the sum:

$$ABS_O(x) = \alpha * \widetilde{A^n}(x) + (1 - \alpha) * \widetilde{\Delta_O^n}(x)$$

This $ABS_O(x)$ value contains both pieces of information: the activity and the objective modifications.

Remarks: The hybridization of many others strategies is as simple as for ABS. For the following sections, we respectively denote the hybridized versions of ABS, IBS and $WDeg$ by ABS_O, IBS_O and $WDeg_O$.

4 Experiments

4.1 The Experimental Setting

Configurations All experiments were done on a Dell machine having four Intel E7-4870 Intel processors and 256 GB of memory, running Scientific Linux. We implemented these new strategies in the Choco 4 CP solver [19]. The code can be found on our GitHub[1]. Each run used a time limit of 30 minutes. The strategies were warmed up with a diving step, using up to 1000 restarts, or by ensuring a certain number of decisions. The same warm up (method and seed) was used for all the methods, in order to avoid any bias.

Benchmarks The experimental evaluation used on the MiniZinc Benchmark library[15], with benchmarks that have been widely studied, often by different communities, including *template design*, *still life*, *RCPSP*, *golomb ruler*, etc. Many problem specifications can be found in [6]. Every class of optimization problems from the MiniZinc library has been considered. Since the number of instances per family is huge, and has a large variance between families, we have randomly selected up to 10 instances per family. Such subset selection preserves the diversity of instances, and do not favor a specific kind of family in plots. The problems have been translated into the FlatZinc format, using the MiniZinc global constraints library provided by Choco-solver, which preserves the global constraints.

Plots The scatters and curves presented in this section are in log scale. A scatter plot shows the comparison of two strategies instance by instance. The diagonal separates the instances where each method has performed better than the other. The points above (resp under) the line correspond to the instances where the ordinate (resp the abscissa) strategy is less efficient. Larger is the gap between the axis line and the point, bigger is the difference between the strategies. Extreme points above and on the right correspond to the timeouts.

Terminology An instance is said to be *solved*, when the best solution has been found and its value proved to be optimal. The term *solution quality* is used when the search is incomplete, and only the best found solution can be judged.

4.2 OBS evaluation

Once again, the *OBS* selector is highly configurable: each bound can have its own coefficient impacting the selection process. The running time of several configurations with different bounds importance have been profiled. The values -1, 1 and 0 have been tested to respectively give : negative, positive, or no importance to the considered bound. All possible pairs of (a,b) from $(-1, 0, 1) \times (-1, 0, 1) \backslash \{(0, 0)\}$ have been tested.

The performance of different *OBS* parameters are shown in Figure 3. This cumulative plot shows how many instances can be solved by each method, for a given time limit. This plot shows that a negative cost to the lower bound outperforms zero or positive cost, regardless of the upper bound.

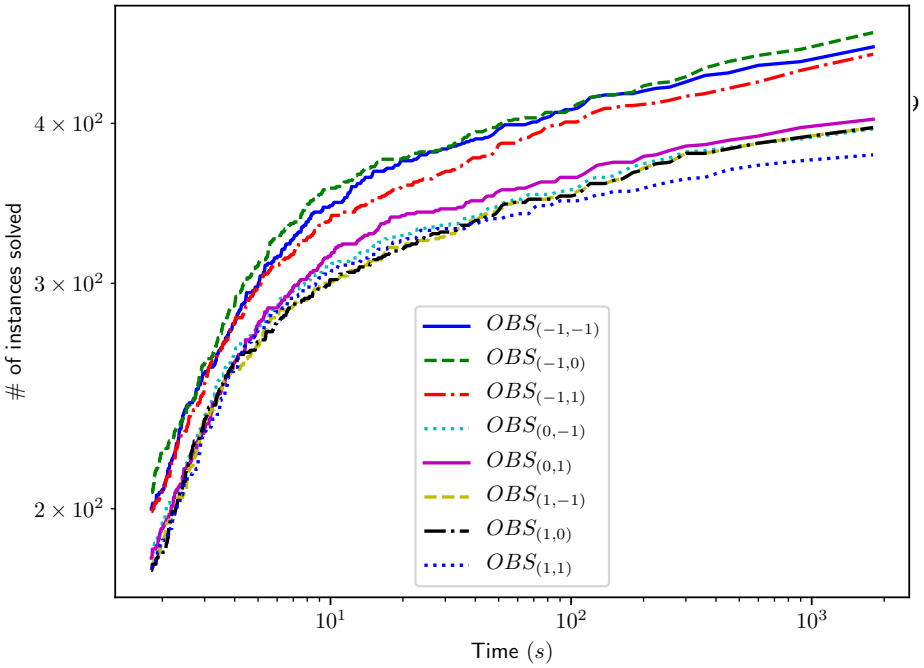

Fig. 3. Comparison of the number of solved instances for different *OBS* configuration.

Configurations weighting the lower bound negatively solve approximately 50 more instances than the alternatives. The solution quality has been compared as well: Figure 4 shows how many time a search has found the best solution (not necessarily optimal) compared to its alternatives. Once again, the searches weighting the lower bound negatively show better results. One intuitive explanation is that choosing the variables impacting the less the bound which has to be optimized, concentrates the search into the most promising parts and like shown in Example 2 helps to back-propagate the objective to prune the tree search. Furthermore, the upper bound in optimization problems (here minimization, without loss of generality) does not have a big impact on resolution time. In addition to our previous intuitions, the upper bound seems to be very sensitive to initialization and to propagation. For instance in some constraints such as *sum*, which often determines the objective value, no arc consistency can be achieved in polynomial time. But, often, only the bounds are filtered, making less consistent the variations of this variable. Based on different OBS experiments, the configurations *(a=-1,b=0)* and *(a=-1,b=-1)* seem to be the most promising.

4.3 Evaluation of Hybrid Strategies

We tried the hybridization with all the *OBS* configurations in order to select the most promising one. The configuration $(a = -1, b = 0)$ got better results within the hybridization, both in run-times and best objectives, which confirms our previous results. In the following, when no configuration is specified for *OBS*, then it means that the configuration $(a = -1, b = 0)$ has been used. Like *OBS*, the hybridization method is

[1] Link

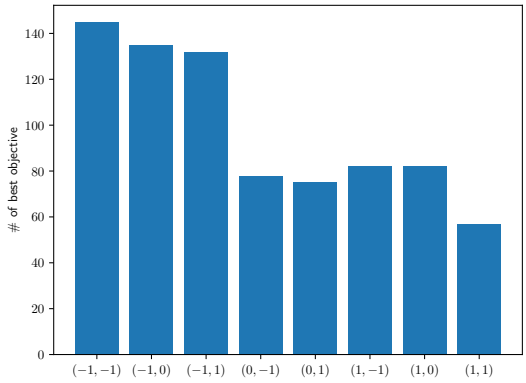

Fig. 4. Comparison of the objective quality between different OBS configurations.

configurable in different ways. More or less importance can be given to the objective, or to the classical feature. In order to find the best parameter α, different experiments have been done. Figure 5 shows the comparison of different values of α on *ABS*. When *OBS* and *ABS* are not hybridized ($\alpha = 1$ and $\alpha = 0$), they clearly show orthogonal behaviors: the timeout are observed on different instances. By looking at the Figure 5, it can be seen that $ABS_O(0.5)$ dominates the others: less timeout and better run-times are observed. Only a full comparison of *ABS* is presented here. We intentionally omit the remaining combinations to preserve clarity, but similar results are observed with the others hybridized strategies. The best combinations are reached when $\alpha = 0.5$. Thus in the following, when we are going to talk about a hybridized version, it will be always with $\alpha = 0.5$.

The run-time and timeout comparisons between the others searches and their hybridized version are shown in Figure 6. It is import to remark that $WDeg_O$ seems to outperform its original version, unlike IBS_O which has an orthogonal behavior. The figure 7 shows how the objective feature impacts the search to find good solutions. It compares the number of times a search against its hybridized version has found a better solution. Unlike *IBS*, *ABS* and *WDeg* seem to benefit from the objective features, since their hybridized versions often find better solution than the original ones. For instance, the classical *ABS* find less than 10 better solution compared to its hybridized version which find more than 100 times.

To support again the interest of the hybridization method, we have extracted some interesting problem families in the Table 1. In this Table, even if *OBS* is not the best strategy, it is often able to solve problems where classical strategies do not. Furthermore, this table shows the interest of the hybridization, which most of the time takes advantage and improve the search considering only one feature. A good example is talent scheduling problem, *OBS* has 8 timeouts and *ABS* 7, but the hybridized version have only 4.

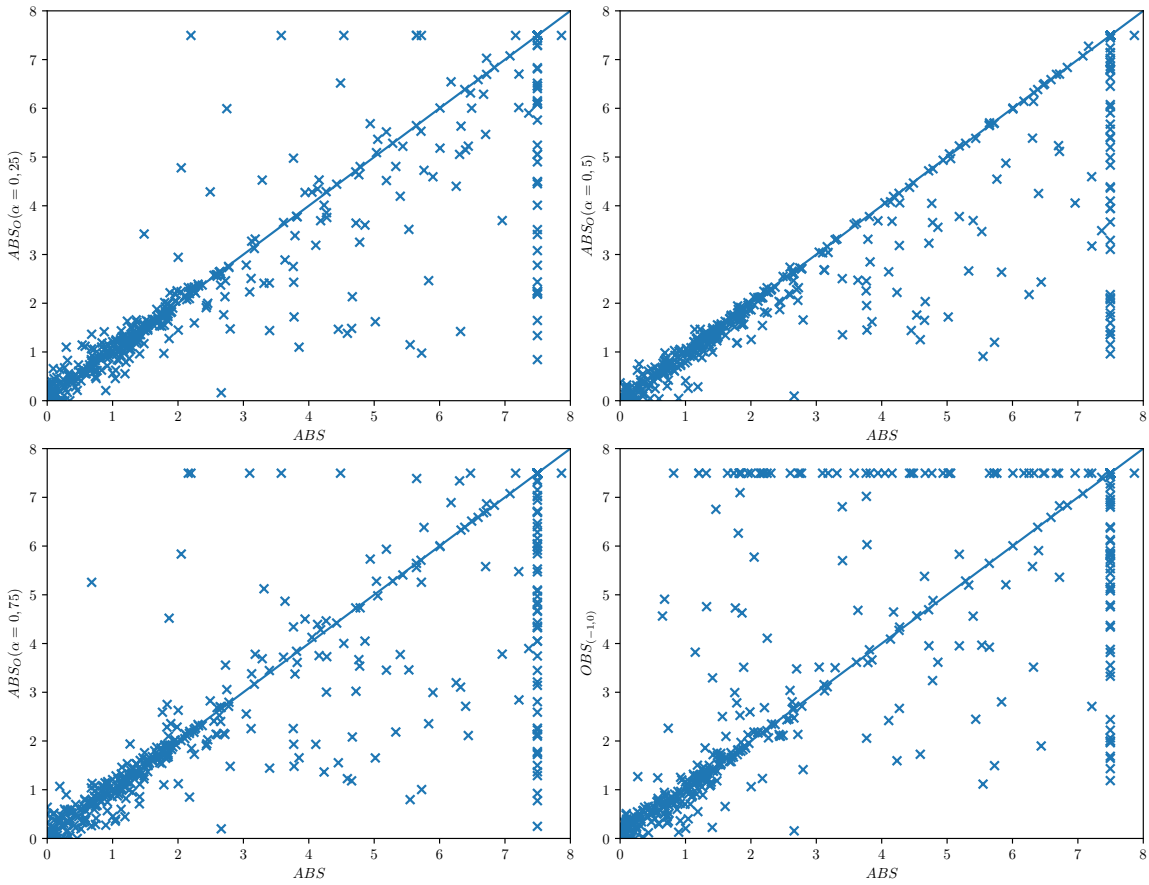

Fig. 5. Running time comparisons of the hybrid strategies, with respect to the hybridization parameters. From left to right and the top to the bottom, the configurations of *ABS* with (0.25), (0.5), (0.75) and (0).

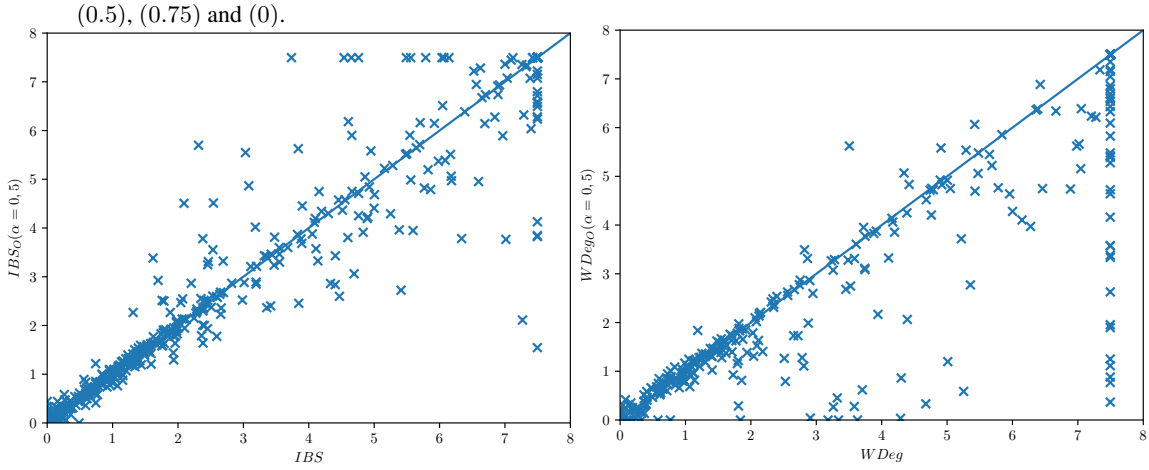

Fig. 6. Comparison of the original search against their hybridized version

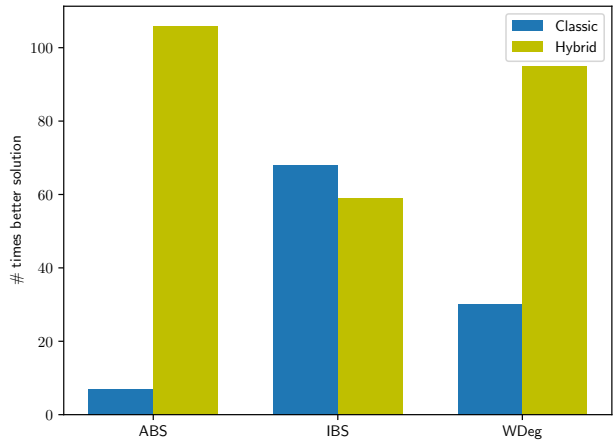

Fig. 7. Comparison of the objective quality between search strategies and their hybridized versions.

Family	*OBS*	*ABS*	*WDeg*	*IBS*	ABS_O	$WDeg_O$	IBS_O
tdtsp	**2**	5	5	5	5	5	5
prize-collecting	**2**	7	7	7	7	**2**	8
2DBinPacking	7	8	8	8	**6**	8	8
mrcpspmm	**0**	3	1	1	**0**	**0**	**0**
mario	**0**	4	2	**0**	4	**0**	3
tpp	7	7	10	10	**5**	10	10
depot placement	3	7	7	**1**	4	6	4
p1f	2	**1**	7	**1**	2	7	3
table-layout	**0**	**0**	10	4	**0**	3	5
filters	2	**1**	5	**1**	**1**	5	**1**
amaze	4	**3**	5	4	**3**	5	4
open stack	8	5	10	7	**4**	9	6
talent scheduling	8	7	8	7	**4**	8	7

Table 1. Number of timeout in some families of instances.

From the different plots and table presented, we remark that *IBS* is an exception because neither the original nor the hybridized version dominates each other and thus does not benefit as much as other search strategies from the hybridization. Actually, *IBS* contains already some information about the objective bounds modifications. The impact is computed over all variables including the objective. This is why the combination of the two features does not lead to a domination, but only an improvement in several problems and a decrease in some others. The resulting search is an orthogonal search to *IBS*.

4.4 Overall evaluation

Figure 9 shows how many instances were solved as a function of time over all strategies. Without any hybridization *IBS* is the best strategy. However, with the hybridization, *ABS* shows the best improvements and so ABS_O become the best strategy. ABS_O has the largest number of solved instances under the allotted time. Furthermore, the hybridized versions are very competitive and improve the number of solved instances. Such a result confirms that using the objective as feature leads to strong improvement in solving time.

Most of the time, in real life problems, the optimal solution cannot be found or proved due to time limits. That is why we now compare the capabilities of *OBS* and the hybridized versions to find good solutions under an allotted time. The new hybrid strategies are very competitive in finding good solutions under a given amount of time as well. Figure 8 shows how many times a search strategy has found a strictly better solution than all the others. Searches using the objective feature are depicted in yellow and the others in blue.

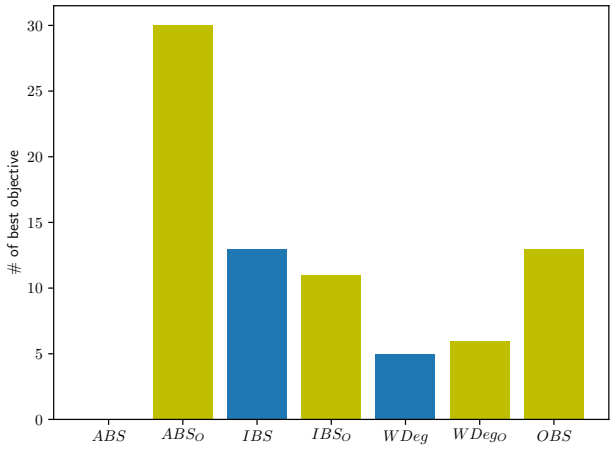

Fig. 8. Number of instances where each search strategy has found a strictly better objective compared to all the other.

ABS_O surpass the others and was able to find 30 times a strictly better objective than the others, while its original version *ABS* never finds a better solution. *IBS* and *OBS* seem to be the second best search strategies in terms of score. The hybridization shows again its advantages since ABS_O is strictly better than ABS. $WDeg_O$ slightly dominates $WDeg$ and *OBS* has a good rank.

Miscellaneous discussions The objective can be monitor in many different ways. The $\Delta_O(t)$ was not our only trial, we tried to monitor the changes through a qualitative function counting how many times a variable modifies either the lower or the upper bounds. On the Minizinc Library, the qualitative function was dominated by the quantitative one.

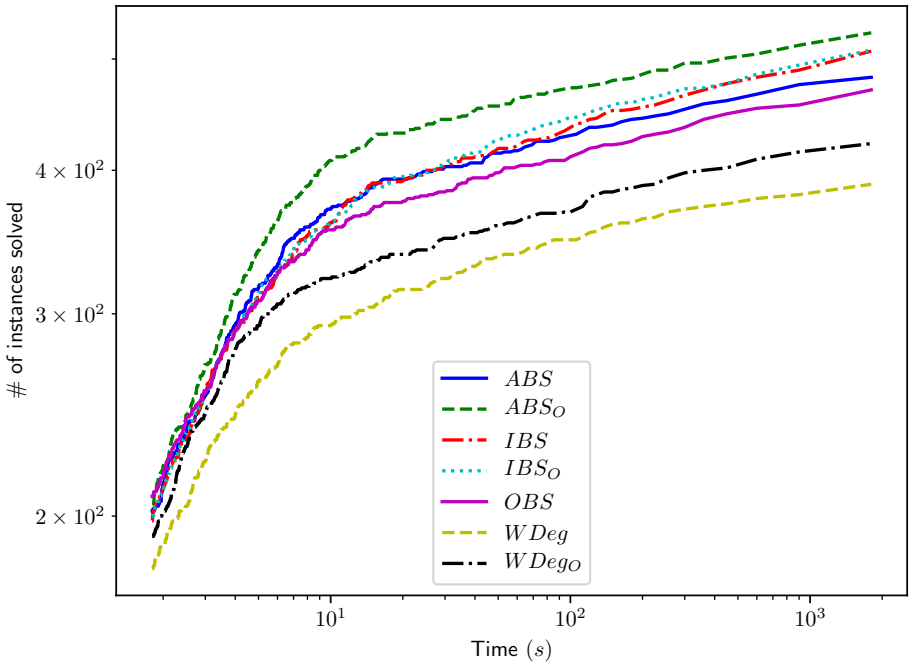

Fig. 9. Comparison of the number of instances solved by the different strategies as a function of time.

Furthermore the $\Delta_O(t)$ function was used to designed a value selector. Different variants have been tried: first to select the value minimizing $\Delta_O(t)$, with possibly different values for a and b. Second to select the value using the new value heuristic from [3]. However, even if on some instances such as *ghoulomb* or *openstack* these selector showed a real improvement, they seem to globally be dominated in the Minizinc problems set by *minVal*. The definition of a good value heuristic seems to still be a challenge to solve.

Our experimental section shows that combining classic search strategies with our objective-based feature leads to better performances and the ability to solve new problems. It shows that for *ABS* and *WDeg* adding an objective-based feature seems to dominate their performance. Finally it shows that the objective as a feature can play an important role in finding a good solution faster, as already claimed [3].

5 Related Work

The objective variable in COPs has already been considered in other fields such as max-SAT [8] to choose which literal to select, or in Soft-CSP for the decision value [10]. Large Neighborhood Search (LNS) framework also consider the objective: for example by changing the term of the weighted sum to minimize [13]. In constraint programming,

the objective information is not yet well used. In [12], the authors propose a heuristic for weighted constraint satisfaction problems based on the solution quality to guide the value selection during the search. In [2], the authors propose a machine learning approach to learn the objective function from the variables' values, but not directly on the variables themselves.

More recently, counting based search has been adapted for optimization problems [17], the main idea being to consider objective-based solution density instead of a simple solution density. This is done by adding to each objective-based constraint an additional algorithm processing these values. Also, in MIP, the objective is widely used in the heuristic [4]: the variables having the best impact on the objective value of the relaxed problem are selected first. This approach differs from our, since CP does not have good relaxation as MIP and we consider the hybridization of the search strategies. A recent work [3] uses the objective information in order to select the variable value, leaving the variable selection to another strategy. Our method differs from [3] since we propose a variables selector, while [3] proposed a value selector. Secondly, we are trying to learn on-the-fly all along the search tree which variable seems to be the most promising, unlike [3]. In [3] the value is selected by testing all the possible assignments of the variable's domains to determine after the propagation which value is the best. Moreover, our feature is more fine-grained because it can be determined how strongly to emphasize bound modifications, using positive or negative parameters. In addition, in this paper we propose an hybridization of existing searches with the objective feature. More particularly our new strategies can be added into the set of available strategies to choose to solve a problem, even in online fashion [27].

6 Conclusion

In this paper we have demonstrated the need for using the objective variable as a feature for decisions within search strategies in constraint programming. We have defined a fine grain feature based on objective bound modifications. By using this new feature, we have designed a new variable selector named OBS. This new variable selector is not the most efficient, but it is able to overpass the existing ones on some class of problems. Moreover, we have proposed a hybridization method to combine our proposed objective-based feature with many existing search strategies. Our evaluation has shown that the hybridized searches give great results and are better than the original strategy in term of run time and solution quality. Some searches are dominated by their hybrid versions. Through this new perspective, we have shown that using the objective as a feature to make decisions can lead to strong results. In addition, further work can be done, for example, with non valued SSs like the ones using a ranking criteria such as COS [5]. Directly applying this work on such SSs is not trivial, and should be a next step.

For both the $\Delta_O(t)$ function and the hybridization, we consider here only a linear combination of the values. More complex combination scheme can be considered. For example, non linear function or ranking function could be studied.

Finally, parameter optimization methods [9] could be used in order to find the best values of a, b and α for a given family of problem while solving it.

References

1. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.
2. Geoffrey Chu and Peter J. Stuckey. Learning value heuristics for constraint programming. In *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, pages 108–123, 2015.
3. Jean-Guillaume Fages and Charles Prud'Homme. Making the first solution good! In *ICTAI 2017 29th IEEE International Conference on Tools with Artificial Intelligence*, 2017.
4. Jean-Michel Gauthier and Gerard Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Math. Program.*, 12(1):26–47, 1977.
5. Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 140–148. Springer, 2015.
6. Ian P Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.
7. Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.
8. Federico Heras and Javier Larrosa. New inference rules for efficient max-sat solving. In *AAAI*, pages 68–73, 2006.
9. Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1209–1216. ACM, 2013.
10. Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
11. Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
12. Nicolas Levasseur, Patrice Boizumault, and Samir Loudni. A value ordering heuristic for weighted csp. In *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*, volume 1, pages 259–262. IEEE, 2007.
13. Michele Lombardi and Pierre Schaus. Cost impact guided lns. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 293–300. Springer, 2014.
14. Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 228–243. Springer, 2012.
15. Nicholas Nethercote, Peter Stuckey, Ralph Becket, Sebastian Brand, Gregory Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543, 2007.
16. Anthony Palmieri, Jean-Charles Régin, and Pierre Schaus. Parallel strategies selection. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 388–404, 2016.
17. Gilles Pesant. Counting-based search for constraint optimization problems. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 3441–3448, 2016.

18. Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res.(JAIR)*, 43:173–210, 2012.
19. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.
20. Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer, 2004.
21. Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Aaai*, volume 94, pages 362–367, 1994.
22. Pierre Schaus, Pascal Van Hentenryck, and Jean-Charles Régin. Scalable load balancing in nurse to patient assignment problems. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 248–262. Springer, 2009.
23. Helmut Simonis and Barry OSullivan. Search strategies for rectangle packing. In *International Conference on Principles and Practice of Constraint Programming*, pages 52–66. Springer, 2008.
24. Barbara M Smith and Stuart A Grant. Trying harder to fail first. *RESEARCH REPORT SERIES-UNIVERSITY OF LEEDS SCHOOL OF COMPUTER STUDIES LU SCS RR*, 1997.
25. Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 437–453. Springer, 2015.
26. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.
27. Wei Xia and Roland H. C. Yap. Learning robust search strategies using a bandit-based approach. In *AAAI Conference on Artificial Intelligence*, 2018.
28. Heytem Zitoun, Claude Michel, Michel Rueher, and Laurent Michel. Search strategies for floating point constraint systems. In *International Conference on Principles and Practice of Constraint Programming*, pages 707–722. Springer, 2017.